2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications

Electro: Toward QoS-Aware Power Management for Latency-Critical Applications

Yanchao Lu*†, Quan Chen*†, Yao Shen*, and Minyi Guo*†
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
†Shanghai Institute for Advanced Communication and Data Science, Shanghai Jiao Tong University, Shanghai, China
Email: chzblych@sjtu.edu.cn, {chen-quan, yshen, guo-my}@cs.sjtu.edu.cn

***Abstract*—Reducing the energy consumption of datacenters is critical for their scalability, sustainability, and affordability when hosting latency-critical applications. Prior work has focused on single-thread applications with a stable workload. Recently, multi-thread latency-sensitive services are widely used in current datacenters. However, the variability of user queries in these service makes existing schemes ineffective, leading to either QoS violations or higher energy consumption. In order to address this new problem, we propose Electro, a machine learning enhanced dynamic power management system. Electro consists of a query duration predictor and a query consolidating engine. The duration predictor can precisely predict the duration of each user query in different scenarios based on the pre-trained duration models. At runtime, according to the predicted duration, the query consolidating engine consolidates user queries accordingly to maximize the duration of the CPU idle states while guaranteeing the QoS. The longer each idle state is, the deeper low-power sleep states can the CPU enter. Our evaluation results on the latest Intel Xeon V4 CPU show that Electro reduces the energy consumption by 81.8% on average compared with the default OS scheduling, and by 14.4% on average compared with the state-of-the-art technique while achieving the 95%-ile latency target for latency-sensitive applications.**

***Index Terms*—Quality of Service; Energy Consumption; Machine Learning; Consolidated Execution;**

I. Introduction

Modern large-scale datacenters host interactive *Latency Critical* (LC) services, such as traditional web service, web search, and the new intelligent personal assistant (IPA) service [1]. Compared with the traditional datacenter services, emerging datacenter services like IPA service [1], DNN service [2], financial service [3], Apache Lucene [4], and Microsoft Bing [5], are more computation demanding, and each user query is processed with multiple threads. While these services require the consistent and instantaneous response for the good user experience, different queries often have different workloads. In order to satisfy the Quality-of-Service (QoS) requirement of the emerging LC services, the resources in a datacenter are often over-provided so that the query that has the heaviest workload can complete within the given QoS threshold. In this case, while the power consumption of emerging datacenters can be tens of megawatts [6], [7], the short queries often have large QoS headrooms because they are processed in a much shorter time than the required QoS target. It is beneficial to trade the extra QoS headrooms for the reduction of the energy consumption.

Quan Chen and Minyi Guo are co-coresponding authors of this paper.

Targeting this problem, prior work has proposed techniques to reduce the power consumption by scaling the processing capability as the workload changes via Dynamic Frequency and Voltage Scaling (DVFS) or Running Average Power Limit (RAPL) [8], [9], [10], [11], [12], [13], [14]. However, these techniques are proposed for the traditional datacenter applications where all the queries have the same workload, and each query is processed by a single thread. They do not apply to emerging multi-thread LC services where the workloads of different queries are not identical and the duration of a query (i.e., the time used to process the query) changes with the number of cores used to process it. On the other hand, these techniques focus on reducing the dynamic power when serving user queries. Recent studies show that [15], [16] static power consumption has become more dominant in emerging multicore processors.

For these reasons, this paper aims to reduce the energy consumption while guaranteeing the required QoS target of multi-thread LC applications through minimizing the static energy consumption. In emerging operating system, when the processors are idle, they enter different levels of sleep states that consumes different static power. The deeper sleep states the processors are in, the lower the static power is. However, it is impractical to control the sleep states directly, because the corresponding control logic is not exposed and wrong states can easily cause serious QoS violation due to the long wake-up overhead. After analyzing the source code of the power management module in the operating system carefully, we find that the processors can enter deeper sleep states that consumes lower static power by increasing the duration of each CPU idle.

According to the above observation, we propose **Electro** runtime system that dynamically consolidates the LC queries that have QoS headrooms and reduces energy consumption through deep sleep states. By consolidating user queries, the frequent and short CPU idles are merged into longer CPU idles. In this way, the processors can enter deeper sleep states that consumes much less power when they are idle. There are three key challenges in Electro. First, different queries have different workloads thus have different QoS headrooms. Previous study shows that the workload of the longest query can be 10X larger than the average one [17]. In this case, a static offline query consolidating method is not applicable.

0-7695-6329-5/17/31.00 ©2017 IEEE
DOI 10.1109/ISPA/IUCC.2017.00040

Second, it is challenging to find the time that each query can be safely delayed without causing the QoS violation due to the complex query queueing behavior. Third, it is challenging to minimize the overall query processing time so that the processors can stay longer in the low-power sleep states.

Electro consists of two parts: a *query duration predictor* and a *query consolidating engine*. The query duration predictor leverages novel models to predict the duration of a query across different inputs and different number of cores. Based on the precise prediction, the query consolidating engine models the query queueing behavior and calculates the time that each query should be launched to the processors, so that the duration of each CPU idle can be maximized while the QoS of the LC services is satisfied. Furthermore, Electro leverages concurrent query execution to minimize the overall time used to process the queries. Our real-system evaluation shows that Electro can significantly reduce the power consumption while guaranteeing the 95%-ile latency of emerging multi-thread LC applications within the QoS target.

The rest of this paper is organized as follows. Section II discusses related works. Section III outlines the design of Electro. Section IV presents the query duration predictor. Section V presents the query consolidating engine. Section VI evaluates Electro and analyzes the experimental results. Section VII draws conclusions.

II. Related work

The diurnal pattern [8], [18], [19] of LC services in datacenters leads to a large amount of prior work on improving resource utilization and computation efficiency while guaranteeing the QoS of LC applications [20], [21], [22], [23]. Work-stealing is one of the mainstream techniques to improve the resource utilization and computation efficiency [24], [25], [26], [27]. These techniques are orthogonal to Electro. They improve the utilization when the system load is high, while Electro reduces the power consumption when the system load is low.

Modern CPUs provide the mechanism for the operating system to scale the frequency and voltage for energy efficiency (DVFS). A great number of prior work has focused on DVFS for better energy efficiency [9], [10], [11]. Anghel et. al [9] developed biology-inspired algorithms to search for the optimal DVFS setting. EEWA [10] tunes the frequencies of individual cores by profiling the performance characteristics of the running task. Kim et. al [11] developed a DVFS performance model to save energy without affecting the performance. However, these studies do not target LC applications with specified deadline. Moreover, prior work [8] has demonstrated that coarse-grain DVFS is ineffective for LC applications.

To fill this gap, several fine-grain power management techniques have been proposed [8], [12], [13]. Adrenaline [13] studied the distribution of the query latency, and proposed a fine-grain DVFS scheme to reduce the tail latency under the same power budget. Rubik [12] developed a performance model and proposed a fine-grain DVFS scheme to meet the sudden changes in the system load. Pegasus [8] utilizes Running Average Power Limit (RAPL) [14], a fine-grain power management mechanism provided in new Intel CPU architecture, to adjust the power limit of processors with the feedback from the last request. While these techniques reduce the dynamic energy consumption through power/voltage scaling, recent work [15], [16] has demonstrated that static power is more dominant in current multicore processors, and saving on static power is more benefitical.

Modern CPUs can enter sleep (low-power) states when they are idle, providing opportunities for reducing the static power. PowerNap [28] forced the processors to enter deep sleep states whenever possible. Although it is energy efficient for the long-run applications, the high overhead of saving CPU states (such as flushing data from cache to memory) makes this technique inefficient and could easily cause QoS violation for LC applications when idles are short [15]. Vacation [29] uses a statistical queuing model to delay the processing of queries, making the idle time longer. However, the model is designed for packet processing, which does not fit the diurnal pattern in datacenters. DynSleep [15] proposed an analytical model to control the tail latency of queries. However, these techniques targets traditional datacenter applications where each query is served with a single thread and different queries have the same workloads. They do not apply to emerging multi-thread LC applications where different queries have different workloads and inputs.

Recently, different architectural designs of computer systems are explored on multi-/many-core processors [30], [31], [32], [33], [34], [35], [36]. Those approaches are complementary to the performance, energy and reliability of computer systems.

III. Design of Electro

Similar to prior work [8], [12], [15], Electro targets Google-style datacenters where a server only hosts a single latency-critical service. Figure 1 presents the design overview of Electro that consists of a *query duration predictor* and a *query consolidating engine*. The *query duration predictor* adopts a low overhead machine learning algorithm to predict a query's duration running with a given number of cores. All the submitted queries are stored in a *ready query pool* managed by Electro. For each query in the ready query pool, the *query consolidating engine* calculates the time that the query should be delayed adopting a query consolidating model, so that the queries are consolidated to run continuously while the QoS of each query can be maintained.

Electro reduces the energy consumption while guaranteeing the QoS of LC applications in the following steps. ❶ When an LC query is submitted, its input is analyzed and passed to the query duration predictor. ❷ Taking the input and the number of cores as the features of the duration model, the duration predictor can precisely predict the execution duration of the query. The number of cores passed to the duration predictor is determined by the number of cores in the computer and the size of the active query pool. ❸ After that, based on the

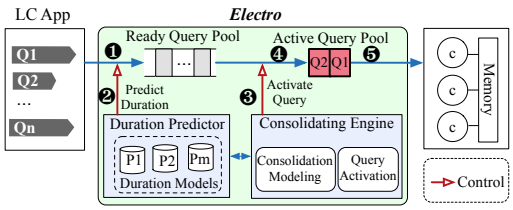


Fig. 1: The design overview of Electro.

predicted duration, the query consolidating engine calculates the time that the query will be activated using the query consolidating model. ❹ Once a query has been delayed for enough time, it is activated and moved from the *ready query pool* into the *active query pool*. ❺ Furthermore, if the active query pool is not empty, the consolidating engine starts all the queries in the active query pool. Because all the queries in the active query pool are executed concurrently, the size of the active query pool is limited to eliminate the possible QoS violation caused by the heavy resource contention between the concurrent queries. By default, we set 3 to the size of the active query pool.

IV. Query Duration Predictor

In this section, we present the modeling methodology used to predict the duration of an LC query.

A. Prediction Methodology

The duration predictor uses the offline-trained duration models to predict the duration of an LC query. When a query is submitted, the duration predictor extracts the representative features, and selects the pre-trained duration model according to the name of the query (the name of the corresponding LC application). Once the duration model is found, Electro predicts the duration of the LC query using the extracted features and the model, and attaches the predicted duration to the query. After that, the query is pushed into the ready query pool waiting to be moved to the active query pool.

Electro builds duration models for each LC application individually for flexibility, because different LC applications often have totally different runtime behaviors and show various performance characteristics. It is challenging to find a performance model that fits all the applications with different characteristics. Besides, if Electro is used to schedule a new LC application, it is much easier to train new performance models for the application independently. On the contrary, if all the applications are modeled in a unified model, the existing model may not fit the new application with an acceptable accuracy. Updating the unified model with the performance data collected from the new application can easily degrade the prediction accuracy of the model for existing applications.

In order to build accurate performance models, we select input data size and the number of cores as the input features because they significantly affect the performance of an LC query. The input data size can be the size of the input files or other abstractions (e.g. number of iterations) based on the application. Because the duration model is trained for each LS application individually, the abstraction of input only needs to be consistent within the same service. We run a large number of LC queries with different inputs on various number of cores for each LC application to collect the training data, using 80% of them as the training set and 20% as the testing set.

B. Selecting Prediction Models

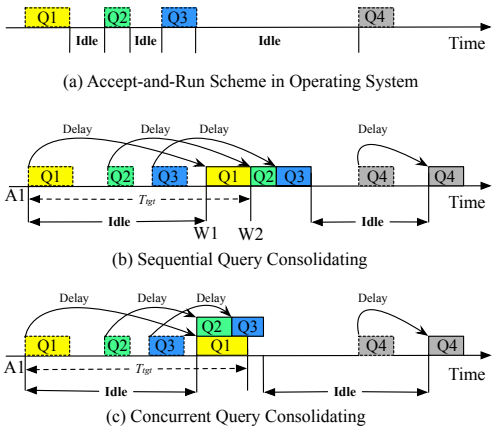
The QoS target of an LC query is often in the granularity of hundreds of milliseconds to support smooth user interaction. Therefore, choosing the modeling techniques with low computation complexity and high prediction accuracy for the online duration predictor becomes critical.

In this work, we evaluated a spectrum of widely used prediction models to predict the query duration. More specifically, we evaluated *K-Nearest Neighbor* (KNN), *Linear Regression* (LR), *Support Vector Machine* (SVM), *Stochastic Gradient Descent* (SGD), and *Multi-layer Perceptron Neural Network* (NN). In these prediction models, LR and SGD assume the linear relationship between input and output variables, while KNN, SVM and NN do not hold such assumption. Our real-system experiment shows that the prediction overhead with these prediction techniques is under $2\,\mathrm{ms}$, that is negligible compared with the $150\,\mathrm{ms}$ QoS target.

In order to achieve high prediction accuracy and high flexibility, we apply all the above prediction models (KNN, LR, SGD, SVM, and NN) to each LC application, and choose the model with the highest accuracy on the testing set to predict the query prediction at run-time. Because the model selection is only done once offline, it does not introduce any extra runtime overhead. Since the duration models are trained offline with the profiled performance samples from the workloads, more sample data is usually effective to improve the accuracy of the duration models. Especially, in datacenters, the workloads become stable after certain time scale and the models become more accurate with periodical updates.

V. Query Consolidating Engine

Electro maximizes the duration of the CPU idle state so that the energy consumption can be minimized. In order to achieve this purpose, the queries could be consolidated and executed continuously as shown in Figure 2(b) and Figure 2(c). Observed from the figure, by consolidating the LC queries and executing them sequentially, the duration of each idle state increases which in turn results in the lower energy consumption. In addition, by allowing concurrent query processing, the overall query processing time is further reduced that in turn increases the duration of the CPU idle state. This is because the time used to processing two queries concurrently is shorter than the overall time used to processing them sequentially [37]. However, the concurrent queries contend for the shared resources and the contention affects the duration of each query. In the following of this section, we first model the sequential query consolidating and then build the concurrent model based on the sequential model.



(a) Accept-and-Run Scheme in Operating System

(b) Sequential Query Consolidating

(c) Concurrent Query Consolidating

Fig. 2: Comparison between the accept-and-run scheme in operating system, sequential query consolidating, and concurrent query consolidating.

A. Modeling Sequential Query Consolidating

In the query consolidating model, the key part is to precisely calculate the time that an LC query has to be activated so that its end-to-end latency would not exceed the QoS target. Let T_{tgt} represent the QoS target of an LC application. Take query Q_1 that is submitted at time A_1 in Figure 2(b) as an example. Suppose its duration predicted with the query duration predictor is D_1. If all the predecessors of query Q_1 have completed before Q_1 is accepted, Eq. 1 calculates the time when Q_1 has to be activated (denoted by W_1) so that its QoS is maintained.

$$W_1 = A_1 + T_{tgt} - D_1 \tag{1}$$

With the sequential query consolidating, when a query is submitted, if its predecessors are not complete, it is executed right after all its predecessors complete. For instance, in Figure 2(b), query Q_2 is activated once Q_1 completes. Equation 2 calculates the time when Q_2 will be activated (denoted by W_2).

$$W_2 = W_1 + D_1 = A_1 + T_{tgt} \tag{2}$$

Without loss of generality, let A and D represent the time when a query Q is accepted, and the predicted duration of Q respectively. Suppose the activate time of its predecessor query $Q_{pre}$is W_{pre} and the predicted duration of Q_{pre} is D_{pre}. Following the same principle in Equation 2, the activate time of query Q, denoted by W, can be calculated by Equation 3.

$$W = \begin{cases} A + T_{tgt} - D & \text{if } A > W_{pre} + D_{pre} \\ W_{pre} + D_{pre} & \text{if } A \leq W_{pre} + D_{pre} \end{cases} \tag{3}$$

However, Equation 3 only considers the query consolidating and neglects the possible QoS violation of Q due to the queueing delay. For instance, if $A \leq W_{pre} + D_{pre}$, it is possible that $W_{pre} + D_{pre} + D > A + T_{tgt}$. In this case, activating query Q at $W_{pre} + D_{pre}$ is too late and results in the QoS violation of query Q. In order to guarantee the QoS of query Q, we update the way to calculate the activate time of query Q to be Equation 4.

$$W = \begin{cases} A + T_{tgt} - D & \text{if } A > W_{pre} + D_{pre} \\ min\{A + T_{tgt} - D, W_{pre} + D_{pre}\} & \text{if } A \leq W_{pre} + D_{pre} \end{cases} \tag{4}$$

Meanwhile, when $A \leq W_{pre} + D_{pre}$ and $W = A + T_{tgt} - D$, it means the activate time of query Q is moved backward for $(W_{pre} + D_{pre}) - (A + T_{tgt} - D)$. Due to the sequential query consolidating, the activate time of each of Q's un-executed predecessors is reduced by $(W_{pre} + D_{pre}) - (A + T_{tgt} - D)$ in consequence.

B. Modeling Concurrent Query Consolidating

The above sequential consolidating method works well for traditional single-thread LC applications, in which the queries are processed sequentially. However, forcing the queries to run sequentially results in the long query processing time. Especially, for emerging multi-thread LC applications, allowing multiple LC queries to run concurrently on the multi-core processor can greatly reduce the overall query processing time. The shorter query processing time means longer low power idle time for the processors. For instance, the makespan of processing Q_1, Q_2 and Q_3 concurrently in Figure 2(c) is shorter than the corresponding makespan of processing them sequentially.

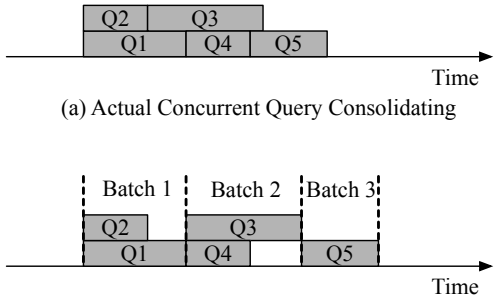
To this end, Electro consolidates the queries that have QoS headrooms and allows the queries to run concurrently. There are two problems have to be solved in the concurrent query consolidating. First, it is hard to predict the duration of a query when it runs concurrently with the other queries. Second, it is hard to determine how many queries should be run concurrently.

1) Predicting the duration of co-running queries: Recall that if a query Q with input data size S running alone on a N-core processor, Electro predicts Q's duration to be $D = predictor(S, N)$. Electro predicts a querys duration at co-location leveraging its solo-run duration model, because the resources are fairly shared between the homogeneous queries by the CFS [38] (Complete Fair Scheduler) scheduler in Linux. If the application is compute-bound, the performance of the query is limited by the number of cores allocated to the query. Therefore, if query Q is compute-bound and runs concurrently with another n queries on the N-core computer. The duration predictor can predict its duration in Equation 5.

$$D = predictor(S, \frac{N}{n+1}) \tag{5}$$

On the other hand, if the application is memory-bound, its performance is limited by the memory bandwidth. When multiple memory-bound queries run concurrently, they contend for the limited memory bandwidth. In the worst case, if a memory-bound query Q runs concurrently with another n queries, its duration can be calculated by Equation 6.

$$D = predictor(S, N) \times n + 1 \tag{6}$$



(a) Actual Concurrent Query Consolidating

(b) Modeled Concurrent Query Consolidating

Fig. 3: Comparison between the actual and the modeled concurrent query consolidating in Electro.

2) Calculating the activate time: The query consolidating engine allows multiple LC queries to run concurrently for reducing the overall query processing time. In order to avoid that the execution time of a query itself exceed the QoS target due to the resource contention, the engine limits the number of queries that can run concurrently. For instance, Figure 3 shows a scenario that Electro allows two LC queries to run concurrently. As shown in Figure 3(a), when a query completes, a query in the ready query pool will immediately start to run. However, computing the activate time is complex in this scenario, defeating the purpose of saving energy. To this end, we model the simplified concurrent query consolidating shown in Figure 3(b). As shown in Figure 3(b), in this model, the query consolidating engine groups the ready queries into batches, and launches the batches to the processor sequentially. By comparing Figure 3(a) and (b), we can find that all the queries in the actual execution complete before their counterparts in the modeled execution. Therefore, if the QoS of the queries can be satisfied in the modeled scenario, their QoS can also be satisfied in the actual execution.

Let k represent the batch size. For a batch b that has k queries, its duration D_b can be estimated by Equation 7. In the equation, D_1, ..., D_k is the predicted duration of the k queries, which are calculated as presented in Section V-B1.

$$D_b = max\{D_1, D_2, ..., D_k\} \tag{7}$$

In concurrent query consolidating, we treat the queries in a batch as a whole. Because the batches are processed sequentially, we can leverage the sequential query consolidating model to predict the activate time of the queries in each batch. For batch b, let A_b and D_b represent the arrive time of the first query in the batch and the predicted duration of batch b. Suppose the activate time and predicted duration of its predecessor batch is W_{pre} and D_{pre}. The activate time of batch b, W_b, can be calculated in Equation 8, deducted from Equation 4.

$$W_b = \begin{cases} A_b + T_{tgt} - D_b & \text{if } A_b > W_{pre} + D_{pre} \\ min\{A_b + T_{tgt} - D_b, W_{pre} + D_{pre}\} & \text{if } A_b \leq W_{pre} + D_{pre} \end{cases} \tag{8}$$

TABLE I: Hardware, software and benchmarks

	Specifications
Hardware	CPU: 2 × Intel Xeon E5-2650 V4 @ 2.20GHz 24 hardware cores, 30MB of L3 cache per socket
OS	CentOS 6.8 x86 64 with kernel 2.6.32-642
Benchmark	**Workloads**
Sirius [1]	dnn-asr (asr), crf, fd, fe, gmm, regex, stemmer (stem)

For the same reason described in Section V-A, when $A_b \leq W_{pre} + D_{pre}$ and $W_b = A_b + T_{tgt} - D_b$, it means the activate time of batch b is moved backward for $(W_{pre} + D_{pre}) - (A_b + T_{tgt} - D_b)$. The activate time of each of Q's un-executed predecessor batches is reduced by $(W_{pre} + D_{pre}) - (A_b + T_{tgt} - D_b)$ in consequence.

Once the activate time of a LC query is calculated, it will be activated at the calculated time by the query consolidating engine. In our current implementation, Electro controls when to issue each LC query to the CPUs and relies upon the Linux kernel to control the power states.

VI. Experimental Evaluation

In this section, we first describe the experimental setup used to evaluate Electro. Then, we evaluate the prediction accuracy of the query duration predictor. After that, we evaluate the performance of Electro in reducing energy consumption while guaranteeing the QoS of LC queries.

A. Experimental Setup

We use a server that has two Intel 12-core Xeon E5-2650 V4 processors to evaluate the performance of Electro. Each socket has 12 hardware cores and the Simultaneous Multi-threading (SMT) technique that delivers two logic cores on each hardware core are enabled. Therefore, the server has 48 logic cores. The detailed setups are summarized in Table I.

As listed in Table I, we use Sirius suite [1] as the representative LC applications to evaluate the performance of Electro. Because Electro targets Google-style datacenters where a server only hosts a single latency-critical service, we evaluate the performance of Electro with homogeneous workload. The concurrent queries are scheduled by the Completely Fair Scheduler [38] in Linux. To construct the training and testing data sets for our duration models in the query duration predictor, we collect a large number of samples, and randomly choose 80% of the samples as the training set. The rest 20% of the samples are used as the testing set. Specifically, each LC application has multiple inputs with different sizes and durations. For KNN model, we choose the number of nearest neighbors to be 5 ($K = 5$). Throughout this section, the QoS is defined as the 95%-ile latency, and the energy consumption are read from the interfaces provided by the RAPL (Running Average Power Limit) mechanism [14].

As mentioned In Section V-B1, Electro predicts the duration of compute-bound and memory-bound LC queries differently when they run with other queries concurrently. The benchmarks in the Sirius suite cover a wide spectrum of applications with different memory bandwidth usage. According to our

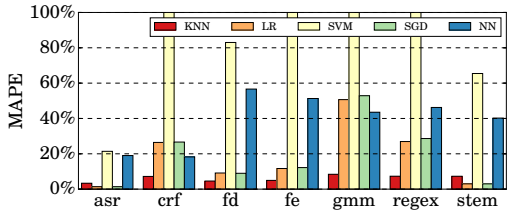


Fig. 4: The prediction error for the duration of the LC applications (lower is more accurate). Either KNN or LR delivers the highest prediction accuracy.

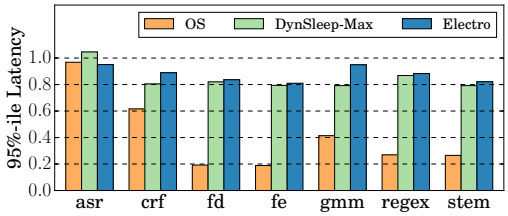


Fig. 5: The normalized 95%-ile latency of the LC applications when scheduled with OS, DynSleep-Max, and Electro.

real-system measurement, *gmm* and *crf* are relatively memory-bound, while the other benchmarks are compute-bound.

B. Accuracy of Duration Prediction

In order to evaluate the prediction accuracy of the above prediction models, we use the *mean absolute percentage error* (MAPE) [39], which is widely-used in evaluating the prediction accuracy in machine-learning field, to evaluate the prediction accuracy of the above modeling techniques for the emerging LC applications. Assume we use a model to predict the duration for n queries of an LC application, Equation 9 calculates the MAPE of the model for the application, where A_i and P_i are the actual duration and predicted duration of a query respectively.

$$M = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{A_i - P_i}{A_i} \right| \tag{9}$$

Figure 4 presents the prediction accuracies of KNN, LR, SVM, SGD and NN for the used Sirius benchmark. Observed from the figure, either KNN or LR performs the best for the LC applications. As we mentioned before, Electro applies all the prediction models to each LC application, and choose the model that fits the data most to predict the query prediction at run-time. Because the model selection is done once offline, it does not introduce any extra runtime overhead. With the flexible offline model selection, Electro selects to use KNN for *crf*, *fd*, *fe*, *gmm*, *regex*, and use LR for *asr*, *stem*. The average prediction error of the selected models in Electro for the LC applications is 5.3% on average.

Besides the prediction accuracy, we also measure the time for making one prediction using each prediction model. According to our measurement on real hardware, the duration prediction overhead with KNN is under $0.4\,\text{ms}$. Meanwhile, the prediction overhead with LR, SVM, SGD, and NN are $2\,\text{ms}$, $2\,\text{ms}$, $0.1\,\text{ms}$ and $9\,\text{ms}$, respectively. Therefore, except NN, the other four prediction techniques have acceptable prediction overhead.

C. QoS and Energy Consumption

In this section, we evaluate the effectiveness of Electro in reducing energy consumption while satisfying the QoS target of emerging multi-thread LC applications. Same to prior work [22], [40], the inter-arrival time is synthetically generated following the exponential distribution with $\beta = 250\,\text{ms}$ so that the query arrive rate follows Poisson distribution.

We compare Electro with the default OS scheduling scheme, and DynSleep [15] in this experiment. With the default OS scheduling, whenever the server accepts an LC query, it starts to process the query immediately for the short end-to-end latency. With DynSleep, similar to Electro, the queries are consolidated and executed sequentially. However, it assumes that all the queries have the same duration and thus is not applicable for the emerging LC applications where different queries have different workloads. In this work, we implement a variation of DynSleep: *DynSleep-Max*. In DynSleep-Max, we use the maximum duration of all the queries to be the unified duration of each query for satisfying the QoS requirement. In this experiment, Electro allows three LC queries to run concurrently.

Figure 5 presents the 95%-ile latency of LC queries when they are scheduled by the operating system, DynSleep-Max scheme, and Electro. For the fairness of comparison, all the schemes run the same query traces, with the same inputs and query arrival intervals. Observed from the figure, both the default OS scheduling and Electro are able to effectively satisfy the QoS for LC applications. On the contrary, DynSleep-Max cannot satisfy the QoS for LC applications (e.g., *asr*). The poor performance of DynSleep-Max for emerging multi-thread LC applications is because they schedule the queries to run sequentially. The sequential query execution results long queuing delay that significantly increases the end-to-end latency of some LC queries. In addition, we can also find that the LC applications have much longer 95%-ile latency in Electro and DynSleep-Max compared with the default operating system scheme. This is mainly because the two schemes delay some LC queries to reduce the energy consumption.

Corresponding to Figure 5, Figure 6 presents the energy consumption of the LC applications when they are scheduled with DynSleep-Max and Electro normalized to their energy consumption with the default OS scheduling scheme. Observed from the figure, DynSleep-Max and Electro are able to greatly reduce the energy consumption of the LC applications compared with OS scheduling, although DynSleep-Max may result in the QoS violation. Electro and DynSleep-Max increase the duration of idle intervals by consolidating LC queries. The longer idle time allows the CPU to enter the deep low-power sleep states. On average, Electro can greatly

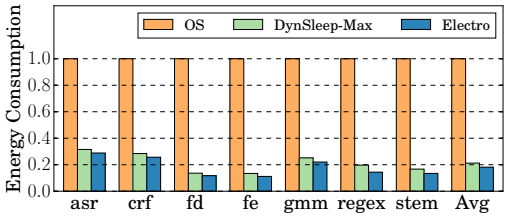


Fig. 6: The normalized energy consumption when the LC applications are scheduled by the OS scheduler, DynSleep-Max, and Electro.

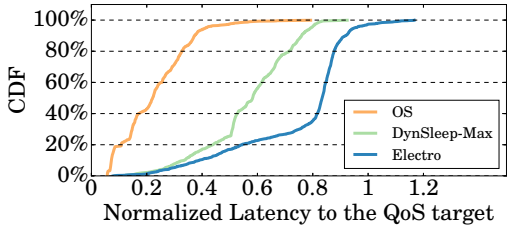


Fig. 7: The cumulative distribution of *gmm*'s end-to-end latency normalized to its QoS target.

reduce the energy consumption by 81.8% compared with the OS scheduling, and by 14.4% compared with DynSleep-Max. Electro increases the duration of idle intervals by consolidating LC queries. The longer idle time allows the CPU to enter the deep low-power sleep states.

In order to show why Electro can greatly reduce the energy consumption, as an example, Figure 7 and Figure 8 present the cumulative distribution of *gmm*'s end-to-end latency normalized to the QoS target, and the cumulative distribution of the CPU idle time when executing *gmm*, respectively. Other benchmarks show the similar results and we omit them due to the limit space.

Observed from Figure 7, the end-to-end latency of more than 80% of the queries with the OS scheduler is shorter than 40% of the QoS target, while more than 80% of the queries with Electro is longer than 80% of the QoS target. The longer end-to-end latency of the queries in Electro results from the query consolidating. In addition, observed from Figure 8, Electro greatly increases the duration of the CPU idles compared with OS scheduler and DynSleep-Max. The longer duration of CPU idles means more aggressive power-down decisions of the Linux kernel. Hence, Electro can enter deeper sleep states more often than OS and DynSleep-Max, saving more energy. This observation is consistent with the finding in prior work [15].

VII. Conclusions

Electro reduces the energy consumption of the system while guaranteeing the QoS target of multi-thread LC applications. Electro delays the execution of queries with an analytical model to increase the duration of CPU's low-power state. The analytical model is enhanced with a machine learning performance model to accurately predict the duration of queries with different inputs at runtime. Moreover, Electro further increases the duration of CPU's low-power state by enabling the concurrent execution of LC queries while guaranteeing the QoS targets. The experimental evaluation shows that Electro reduces the energy consumption by 81.8% on average compared with the default OS scheduling and by 14.4% on average compared with the state-of-the-art technique, while achieving the 95%-ile latency target for emerging latency-sensitive services.

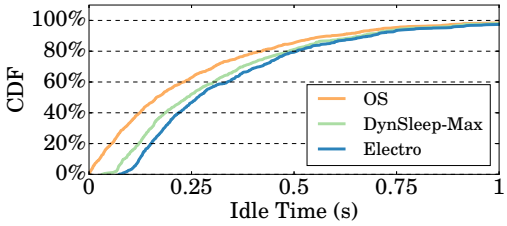


Fig. 8: The cumulative duration distribution of CPU idles when hosting *gmm*.

Acknowledgment

We thank our anonymous reviewers for their feedback and suggestions. This work was partially sponsored by the National Basic Research 973 Program of China under grant 2015CB352403, the National Natural Science Foundation of China (NSFC) (61602301, 61632017).

References

[1] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. Istanbul, Turkey: ACM, 2015, pp. 223–238.

[2] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. Portland, Oregon, USA: ACM, 2015, pp. 27–40.

[3] Z. C. LLC, M. G. Limited, and G. A. M. LLC, "Parallelizing a computationally intensive financial r application with zircon technology," in *Proceedings of the 2010 R User Conference*. Gaithersburg, Maryland, USA: IEEE, 2010, pp. 1–8.

[4] APACHE SOFTWARE FOUNDATION. (2004) Apache lucene. [Online]. Available: https://lucene.apache.org/

[5] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Adaptive parallelism for web search," in *Proceedings of the 8th ACM European Conference on Computer Systems*. Prague, Czech Republic: ACM, 2013, pp. 155–168.

[6] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[7] D. Meisner and T. F. Wenisch, "Dreamweaver: Architectural support for deep sleep," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. London, England, UK: ACM, 2012, pp. 313–324.

[8] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. Piscataway, NJ, USA: IEEE, 2014, pp. 301–312.

[9] I. Anghel, T. Cioara, I. Salomie, G. Copil, D. Moldovan, and C. Pop, "Dynamic frequency scaling algorithms for improving the cpu's energy efficiency," in *Proceedings of 2011 IEEE 7th International Conference on Intelligent Computer Communication and Processing*. Cluj-Napoca, Romania: IEEE, 2011, pp. 485–491.

[10] Q. Chen, L. Zheng, M. Guo, and Z. Huang, "Eewa: Energy-efficient workload-aware task scheduling in multi-core architectures," in *Proceedings of 2014 IEEE International Parallel Distributed Processing Symposium Workshops*. Phoenix, AZ, USA: IEEE, 2014, pp. 642–651.

[11] S.-g. Kim, H. Eom, H. Y. Yeom, and S. L. Min, "Energy-centric dvfs controlling method for multi-core platforms," *Computing*, vol. 96, no. 12, pp. 1163–1177, 2014.

[12] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 48th International Symposium on Microarchitecture*. Waikiki, Hawaii, USA: ACM, 2015, pp. 598–610.

[13] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture*. Burlingame, CA, USA: IEEE, 2015, pp. 271–282.

[14] INTEL, INC., *Intel 64 and IA-32 Architectures Software Developers Manual*, 2013.

[15] C.-H. Chou, D. Wong, and L. N. Bhuyan, "Dynsleep: Fine-grained power management for a latency-critical data center application," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. San Francisco Airport, CA, USA: ACM, 2016, pp. 212–217.

[16] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *IEEE Computer*, vol. 36, no. 12, pp. 68–75, 2003.

[17] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[18] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[19] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah, "Worth their watts?-an empirical study of datacenter servers," in *Proceedings of 2010 IEEE the 16th International Symposium on High Performance Computer Architecture*. Bangalore, India: IEEE, 2010, pp. 1–10.

[20] M. Guo, Y. Pan, and Z. Liu, "Symbolic communication set generation for irregular parallel applications," *The Journal of Supercomputing*, vol. 25, no. 3, pp. 199–214, 2003.

[21] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. Tel-Aviv, Israel: ACM, 2013, pp. 607–618.

[22] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge, UK: IEEE, 2014, pp. 406–418.

[23] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144.

[24] Q. Chen, M. Guo, and Z. Huang, "Cats: cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*. San Servolo Island, Venice, Italy: ACM, 2012, pp. 163–172.

[25] Q. Chen, M. Guo, and H. Guan, "Laws: locality-aware work-stealing for multi-socket multi-core architectures," in *Proceedings of the 28th ACM International Conference on Supercomputing*. Munich, Germany: ACM, 2014, pp. 3–12.

[26] Q. Chen and M. Guo, "Adaptive workload-aware task scheduling for single-isa asymmetric multicore architectures," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 1, pp. 8:1–8:25, 2014.

[27] Q. Chen, M. Guo, and Z. Huang, "Adaptive cache aware bitier work-stealing in multisocket multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2334–2343, 2013.

[28] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. Washington, DC, USA: ACM, 2009, pp. 205–216.

[29] C.-H. Chou and L. N. Bhuyan, "A multicore vacation scheme for thermal-aware packet processing," in *Proceedings of 2015 33rd IEEE International Conference on Computer Design*. New York, NY, USA: IEEE, 2015, pp. 565–572.

[30] M. Dong, K. Ota, L. T. Yang, A. Liu, and M. Guo, "Lscd: A low-storage clone detection protocol for cyber-physical systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 712–723, 2016.

[31] C. Li, X. Li, R. Wang, T. Li, N. Goswami, and D. Qian, "Chameleon: Adapting throughput server to time-varying green power budget using online learning," in *Proceedings of 2013 IEEE International Symposium on Low Power Electronics and Design*. Beijing, China: IEEE, 2013, pp. 100–105.

[32] C. Li, Y. Hu, L. Liu, J. Gu, M. Song, X. Liang, J. Yuan, and T. Li, "Towards sustainable in-situ server systems in the big data era," in *Proceedings of 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*. Portland, OR, USA: IEEE, 2015, pp. 14–26.

[33] W. Zheng, K. Ma, and X. Wang, "Te-shave: Reducing data center capital and operating expenses with thermal energy storage," *IEEE Transactions on Computers*, vol. 64, no. 11, pp. 3278–3292, 2015.

[34] M. Guo, E. Olule, G. Wang, and S. Guo, "Designing energy efficient target tracking protocol with quality monitoring in wireless sensor networks," *The Journal of Supercomputing*, vol. 51, no. 2, pp. 131–148, 2010.

[35] G. Wang, H. Wang, J. Cao, and M. Guo, "Energy-efficient dual prediction-based data gathering for environmental monitoring applications," in *Proceedings of 2007 IEEE Wireless Communications and Networking Conference*. Kowloon, China: IEEE, 2007, pp. 3513–3518.

[36] K. Chi, X. Jiang, S. Horiguchi, and M. Guo, "Topology design of network-coding-based multicast networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 5, pp. 627–640, 2008.

[37] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. Porto Alegre, Brazil: ACM, 2011, pp. 248–259.

[38] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, no. 184, p. 4, 2009.

[39] J. S. Armstrong and F. Collopy, "Error measures for generalizing about forecasting methods: Empirical comparisons," *International Journal of Forecasting*, vol. 8, no. 1, pp. 69–80, 1992.

[40] S. Homsi, S. Liu, G. Chaparro-Baquero, O. Bai, S. Ren, and Q. Gang, "Workload consolidation for cloud data centers with guaranteed qos using request reneging," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–14, 2016.